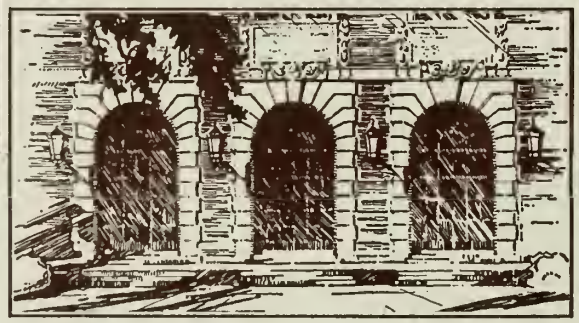

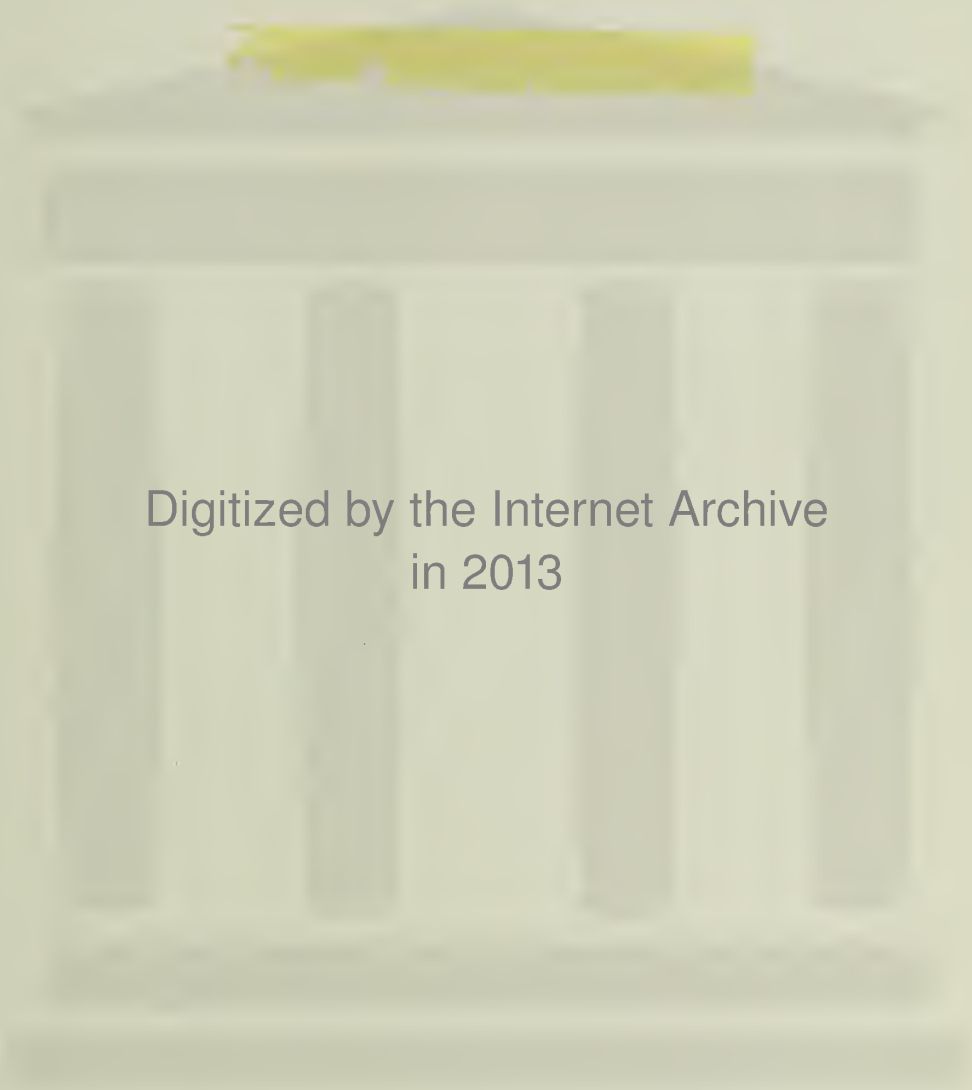

Report No. 218

A LINEAR PROGRAMMING CODE TO DETERMINE WHETHER A BOOLEAN FUNCTION IS A THRESHOLD FUNCTION

by

Charles R. Baugh

January 6, 1967

DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

Report No. 218

A LINEAR PROGRAMMING CODE TO DETERMINE WHETHER A BOOLEAN FUNCTION IS A THRESHOLD FUNCTION

by

Charles R. Baugh

January 6, 1967

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

This work was submitted in partial fulfillment for the degree of Master of Science in Electrical Engineering under the direction of Professor S. Muroga, January 1967.

ACKNOWLEDGMENT

The author wishes to express his appreciation to Professor S. Muroga, Department of Computer Science, who graciously offered many suggestions and comments. His advice was of great benefit to the preparation of this thesis.

Thanks are also expressed to Dr. T. Tsuboi without whose close cooperation the programming of the linear programming code would have been considerably more difficult.

TABLE OF CONTENTS

INTRODUCTION

An interesting subclass of Boolean functions are threshold functions. Muroga, Toda, and Takasu devised a test which incorporated linear programming techniques to determine whether a switching function is a threshold function; and if so, its structure.[1*] If we consider only positive self-dual threshold functions of exactly nine variables excluding functions with permuted variables, we can be assured of discovering all eight variable threshold functions. The advantage of dealing with these self-dual threshold functions is that as many as 18 eight variable threshold functions can be enumerated from one of these nine variable functions.[2]

Thus the problem is to determine all nine variable self-dual threshold functions. Through various reduction techniques[2] the set of all nine variable Boolean functions is reduced to 319,124 possible self-dual threshold functions. With such a large number of functions to check, it is necessary to have a very fast and efficient linear programming code.

A set of inequalities characterize each function and are of the following form:

---

*See List of References.

$$
\begin{aligned}
a_{1,1}'w_1' + a_{1,2}'w_2' + \cdots &\qquad + a_{1,9}'w_9' \geq 1 \\
&\vdots \\
a_{m,1}'w_1' + \cdots &\qquad + a_{m,9}'w_9' \geq 1 \\
w_1' - w_2' &\qquad \geq 0 \\
w_2' - w_3' &\qquad \geq 0 \\
&\ddots \\
&\qquad w_8' - w_9' \geq 0 \\
&\qquad w_9' \geq 0
\end{aligned}
\tag{1}
$$

where $a_{ij}' = \pm 1$ for all i and j.

With the nine variable functions m ranges from 1 to 23 inclusive.* These constraints along with the object function of

$$\min(w_1' + w_2' + \cdots + w_9') \tag{2}$$

constituted the linear programming problem. However, Gabelman[3] and Winder[4] reformulated the problem by substituting w_i for $w_i' - w_{i-1}'$ for $i = 1,\ldots,8$ and w_9 for w_9'. Using this substitution technique, the problem becomes the following:

$$\min(w_1 + 2w_2 + 3w_3 + \cdots + 9w_9) \tag{3}$$

while satisfying

---

* Our programming results yield 23 as maximum m.

$$
\begin{aligned}
&a'_{11}w_1 + (a_{11}' + a_{12}')w_2 + \ldots + (a'_{11} + a'_{12} + \ldots + a'_{19})w_9 \geq 1 \\
&\quad\vdots \\
&a'_{m1}w_1 + (a_{m1}' + a_{m2}')w_2 + \ldots + (a'_{m1} + a'_{m2} + \ldots + a'_{m9})w_9 \geq 1 \, .
\end{aligned}
\tag{4}
$$

Note that the last nine inequalities of (1)* are simply $w_i \geq 0$ for $i = 1,\ldots,9$ and will not be explicitly present in the simplex method. For clarity let $a_{ij} = \sum_{k=1}^{j} a_{ik}'$ with (4) becoming

$$
\begin{aligned}
&a_{11}w_1 + \ldots + a_{19}w_9 \geq 1 \\
&\quad\vdots \\
&a_{m1}w_1 + \ldots + a_{m9}w_9 \geq 1
\end{aligned}
\tag{5}
$$

and $w_i \geq 0$ for $i = 1,\ldots,9$.

Since all of the coefficients of the object function (3) are non-negative and the inequalities of (5) are all in the "greater than or equal" form, the dual of the problem is:

$$\max(v_1 + v_2 + \ldots + v_m) \tag{6}$$

while satisfying

$$
\begin{aligned}
&a_{11}v_1 + \ldots + a_{m1}v_m \leq 1 \\
&a_{12}v_1 + \ldots + a_{m2}v_m \leq 2 \\
&\quad\vdots \\
&a_{19}v_1 + \ldots + a_{m9}v_m \leq 9
\end{aligned}
\tag{7}
$$

---

*These inequalities are useful for reducing the number of inequalities in the first m inequalities of (1). In Winder's formulation, this advantage was not fully used.

where $v_i \geq 0$ for $i = 1,\ldots,m$.

The linear programming problem in this form has an initial feasible solution which eliminates the need for artificial variables and reduces the amount of computation and as a result reduces the time necessary to arrive at the solution. Also when this dual problem is solved by the simplex method, the solution of the primal problem as well as a solution to the dual problem is obtained. In other words, the values of $w_1,\ldots,w_9$ are located in their respective positions of $b_{m+1,10} \cdots b_{m+9,10}$ of (8a).

For clarity write the simplex tableau as:

			1 . . . 1	0 . . . 0
			1 . . . m	m+1 . . . m+9
B_1	C_{B1}	b_{01}	$b_{11} \cdots b_{m1}$	$b_{m+1,1} \cdots b_{m+9,1}$
			. . .	. . .
B_9	C_{B9}	b_{09}	$b_{19} \cdots b_{m9}$	$b_{m+1,9} \cdots b_{m+9,9}$
		$b_{0,10}$	$b_{1,10} \cdots b_{m10}$	$b_{m+1,10} \cdots b_{m+9,10}$

(8a)

Thus the initial tableau is:

			1 . . . 1	0 . . . 0
			1 . . . m	m+1 . . . m+9
m+1	0	1	$a_{11} \cdots a_{m1}$	1
.	0	2		1 ○
.		.	.	.
.		.	.	.
		.	.	.
		.	.	.
m+9	0	9	$a_{19} \cdots a_{m9}$	○ 1
		0	-1 . . . -1	0 . . . 0

(8b)

An iteration of the simplex method is performed by transforming each tableau entry. The conventional transformation of the tableau entries b_{ij} of (8a) is formulated by using a column of (8a) as a reference column $\vec{y}$ and an entry of $\vec{y}$ as a pivot y_ℓ; i.e.

$$b_{ij} \text{ is replaced by } b_{ij} - b_{\ell j} y_i / y_\ell \text{ for } i=0,\ldots,i-1,i+1,\ldots,m+9 \text{ and } j=1,\ldots,10 \tag{9}$$

and $b_{\ell j}$ is replaced by $b_{\ell j}/y_\ell$ for $j=1,\ldots,10$.

Or rewriting we get

$$b_{ij} = [b_{ij} y_\ell - b_{\ell j} y_i]/y_\ell \text{ for } i=0,\ldots,i-1,i+1,\ldots,m+9 \text{ and } j=1,\ldots,10 \tag{10}$$

and $b_{\ell j} = b_{\ell j}/y_\ell$ for $j=1,\ldots,10$.

COMPUTER IMPLEMENTATION

NICAP, ILLIAC II assembly language, is the language in which the linear program code was written. The features of this machine which are particularly appealing are its word structure and control unit. The word is broken into a 44 bit mantissa and a 7 bit exponent which makes ILLIAC II very suitable for numerical problems.

The central control unit is divided into an advanced control which handles indexing functions and delayed control which supervises the accumulator. The advantage of the dual control is that one can be executing an advanced control and delayed control instruction simultaneously. Therefore one can increase the speed of the code by appropriate intermixing of advanced and delayed control instructions. Also the delayed control execution of multiplication and division is bypassed if the accumulator contains zero. Since the problem often contained zeros, the computation time was reduced.

The last feature is the bank of fast memory in which intermediate arithmatic results could be stored and retrieved very quickly. The typical advanced control instruction including store and access of fast memory is 1 μ second. Typical timing for core access is 1.8 μ seconds, for multiplication 6.6 μ seconds, and for addition 3.3μ seconds.

Making use of the fact that the coefficients of the inequalities of (7) are all integers, we were able to eliminate time consuming divisions which introduce and propagate round off error. This is done by putting all entries of the tableau over a common denominator y_ℓ and

then keeping the numerators in the tableau and the denominator in a separate location.

This procedure necessitated the use of the transformation formulae of (10) rather than (9). In (10) we can keep the denominator y_ℓ separate. Thus the denominator at any given iteration of the simplex method is the product of each of the y_ℓ's of the previous iterations.

However, if no provision is made to scale down the denominator and the tableau entries, accumulator overflow will occur since the represented number will be the ratio of two very large members. From the expressions for the numerator entries of the transformed tableau (10), one can see that the magnitude is a function of the entries of the column $\vec{y}$ which contain the pivot. Thus keeping the $\vec{y}$ entries as small as possible yet still integer will prohibit the b_{ij} entries from growing as rapidly. The following simple pseudo common divisor routine was incorporated to prevent overflow and to keep the increased computation time small. First the value of the smallest, in absolute value, of the m+1 $\vec{y}$ column entries, y_k was found. The largest in absolute value of y_k, $y_k/2$, $y_k/3$, $y_k/4$, $y_k/5$ which was both an integer and a common divisor of all $\vec{y}$ column entries was used as the greatest common divisor of the $\vec{y}$ column. By this scheme the mantissa of the tableau entries never became larger than the 44 bits and the magnitudes never exceeded the capacity during the computation.

A consideration of the number of multiplications per iteration needed for the simplex method and the revised simplex method will show which is superior for our class of problems. For the simplex method

there are 10(m+9)+9 entries in the tableau with 2 multiplications for each entry. This amounts to 20m + 198 multiplications. For the revised simplex method the number of multiplications is: 10m to calculate the (z-c)'s of the non-basic variables; 90 to calculate the pivot column entries; and 220 to transform the basis for a total of 10m + 310. The following table shows the comparison of the two methods.

Number of Multiplications for One Iteration for a Problem With m Constraints

	SIMPLEX	REVISED SIMPLEX
m	20m + 198	10m + 310
2	238	330
3	258	340
4	278	350
5	298	360
6	318	370
7	338	380
8	358	390
9	378	400
10	398	410
11	418	420
12	438	430
13	458	440
14	478	450
15	498	460
16	518	470
17	538	480
18	558	490
19	578	500
20	598	510

Thus the revised simplex method is more efficient when m is greater than 11.

In our problems m averaged 4.91 which is well below the approximate breakeven point of 11.2. However, since it would be

convenient to do some post-linear programming operations on the coefficients, the revised simplex method was chosen so that the coefficient matrix would remain unchanged.

With this code we solved 319,124 linear programming problems in 11 hours and 34 minutes with 1,566,704 iterations using ILLIAC II. Thus the average problem took 2.17 m seconds with each iteration consuming 442 μ seconds. This total time includes some pre- and post-linear programming operations which consumed only about 10% of the time. The linear programming code itself used 280 locations of instructions and 262 locations for data storage.

This thesis is a part of the joint research work with Professor S. Muroga and Dr. T. Tsuboi. The problem statement of the joint research work and the results produced from the computer program of this thesis will be published elsewhere.

PROGRAM DESCRIPTION

PROBLEM: LNPG1 is a linear program which

$$\max \sum_{i=1}^{n} c_i x_i$$

while satisfying

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \geq \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \qquad (11)$$

and $x_i \geq 0$ for $i = 1,\ldots,n$

where $n \leq 23$ and $m \leq 9$. All of the real numbers c's, a's, and b's must be integers. The revised-simplex method as outlined in Linear Programming by George Hadley is the method employed.[5] The procedure requires that [A] contain a unit matrix in the last m columns or that a full set of artificial variables be appended.

DATA FORMAT: The coefficients $a_{i,j}$ and constants b_i and c_i must be in one area in core memory and in the following form:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} & b_1 \\ \vdots & & & \\ a_{m,1} & \cdots & a_{m,n} & b_n \\ c_1 & & c_n & 0 \end{bmatrix} \qquad (12)$$

The entries of the matrix in (12) must be stored by row in sequential core memory locations. At the completion of the linear programming calculations the entries of matrix in (12) will be unchanged.

MEMORY DESTROYED: F0, F1, F2, F3, M3, and accumulator.

PROGRAM LENGTH: 280_{10} or 428_{8}

COMMON MEMORY: None

ERASABLE MEMORY: 262_{10} or 406_{8}

OUTPUT: The program calls an external subroutine IØ which must be supplied by the user. The IØ routine is called after every iteration of both Phase I and Phase II of the revised simplex method. The user must return control back to the program by a JLH M3 with all modifiers except M3 unchanged. The user will be given the following information:

M3: Return jump

M4: Count of Phase I iterations

M5: Count of Phase I plus Phase II iterations

M8: Number of equalities in (11)

M9: Number of variables in (11) plus one

M10: Location of $a_{1,1}$ of (12)

M11: First location of basis (stored by column)

M12: First location of the two additional equalities necessary for the revised simplex method (stored by row)

M13: 13th bit (sign bit) is 1 if Phase I is completed, 12th bit is 1 if Phase II is completed, 11th through 1st bit are the same as the user's M11 when LNPG1 was originally called.

M15: Location of denominator of all basis entries.

NO SOLUTION: If the linear program discovers that the problem has no finite solution or no feasible solution, then it calls a subroutine called ERØR. This subroutine must be supplied by the user and control need not be returned to LNPG1. If control is returned to LNPG1 by a JLH M3, it will return control to the user's routine which originally activated LNPG1. The user is given the same information as he was in subroutine IØ with the addition of

M1: is 0 if no feasible solution and is -1 if no finite solution.

All other modifiers are the same as they were when the call to LNPG1 was made.

ACTIVATION OF LNPG1: The modifiers must contain the following information:

M8: Number of equalities of (11)

M9: Number of variables plus one of (11)

M10: Location of a_{11} of (12)

M14: Negative if no artificial variables added;
Positive if full set of artificial variables added.

All other modifiers can contain any information the user desires. The call to LNPG1 is by a

```
CALL LNPG1
```

statement in NICAP.

LIST OF REFERENCES

1. Muroga, Saburo; Toda, Iwao; Takasu, Satoru. "Theory of Majority Decision Elements," Journal of the Franklin Institute, Vol. 271, No. 5, May 1961.

2. Muroga, Saburo. Lecture Notes for Threshold Logic Course, Department of Computer Science, University of Illinois, 1966.

3. Gabelman, I. J. "The Functional Behavior of Majority Elements," Ph.D. Dissertation, Electrical Engineering Department, Syracuse University, Syracuse, New York, 1961.

4. Winder, R. O. "Enumeration of Seven-Argument Threshold Functions," IEEE Transactions on Electronic Computers, Vol. EC-14, No. 3, June 1965.

5. Hadley, George. Linear Programming, Addison Wesley, 1962.

APPENDIX

SUBROUTINE DESCRIPTIONS

LNPG1 first initializes itself and then jumps to an internal subroutine called LP which is the main control of the revised-simplex method. Thus LNPG1 initializes the basis and establishes the two additional constraints for Phase I and Phase II of the revised-simplex procedure as outlined in Chapter 7 of Linear Programming by George Hadley.[5]

SUBROUTINES: (Constitute LNPG1)

INTERNAL

- LP
- FASI
- FASII
- FASIIA
- BXA
- TRANS

EXTERNAL (Supplied by user)

- IØ
- ERØR

DATA: The basis data array includes: (1) the subscripts of the variables which make up the present basis; (2) the inverse of the matrix formed by the columns of (11) which correspond to the variables constituting the basis; (3) the values of these basic variables; and (4) the control

column $\vec{y}$ which corresponds to the last variable introduced into the basis. The basis is stored by column and has the following format:

$$[B \mid \vec{x} \mid \vec{y} \mid \#'s] = \left[\begin{array}{ccc|c|c|c|c} b_{01} & & b_{0,m+1} & x_0 & y_0 & \multirow{4}{*}{\shortstack{Subscript of variables stored in 3rd qt-word}} \\ b_{11} & & & x_1 & y_1 & \\ \cdot & & & \cdot & \cdot & \\ \cdot & & & \cdot & \cdot & \\ b_{m+1,1} & \cdots & b_{m+1,m+1} & x_{m+1} & y_{m+1} & \end{array}\right] \qquad (13)$$

Initially (first basis);

(a) $x_o = 0$, $x_1 = -\sum_{i=1}^{m} b_i$ and $x_i = b_{i-1}$ $\quad i = 2,\ldots,m+1$;

(b) variable numbers correspond to the last m subscripts of the variables in (11) or to n+1 through n+m if artificial variables are added;

and (c) the first m+1 columns of (13) are $\begin{bmatrix} \vec{0}' \\ I_{m+1} \end{bmatrix}$ where $\vec{0}' = [0,\ldots,0]$ and I_{m+1} is a unit matrix of rank m+1.

The coefficients for the additional two rows of (11) necessary for the revised simplex method are stored sequentially by row in an array A12 as follows:

$$\begin{bmatrix} -c_1 & -c_2 & \cdots & -c_m & 0 \\ -\sum_{i=1}^{m} a_{1i} & -\sum_{i=1}^{m} a_{2i} & \cdots & -\sum_{i=1}^{m} a_{ni} & 0 \end{bmatrix} = [A12] \qquad (14)$$

Both (13) and (14) are data arrays in erasable memory. Therefore care must be exercised if the user is also using erasable storage.

LP: Determines whether a full set of artificial variables are needed and branches to Phase I or Phase II depending upon the necessity of artificial variables. Upon the completion of Phase I it enters Phase II. If, however, a feasible solution does not exist, control is transferred to subroutine ERØR which has been supplied by the user. At the completion of Phase II control is returned to the user's calling routine. If an unbounded solution is encountered control is given to the user's subroutine ERØR once again. Control is transferred to user's subroutine IØ at the end of each iteration of both Phase I and Phase II.

FASI, FASII, and FASIIA: This routine with its three entry points FASI, FASII, and FASIIA calculates the variable which is to be introduced into the basis. If a feasible solution does not exist, a flag is set so that LP can detect the condition. If Phase I is completed another flag is set for LP.

FASII serves the same purpose as FASI except that it deals with Phase II instead of Phase I. Upon completion of Phase II a flag is set to signal LP. If the solution is unbounded another flag is set for LP.

Since FASI and FASII are so similar, the main work of both of these routines is handled by FASIIA.

BXA: This short routine conducts the multiplication of a row of the basis of (13) with the column of the modified coefficient matrix $\begin{bmatrix} A12 \\ A \end{bmatrix}$ which consists of (11) and (14). However, the product of $b_{i,o}$ with the first entry of $\begin{bmatrix} A12 \\ A \end{bmatrix}$ is not included in the product. The product is

$[b_{i1}\ b_{i2}\ \cdots\ b_{i,m+1}]$ with the second through the last entries of the column of $\begin{bmatrix} A12 \\ A \end{bmatrix}$.

TRANS: The routine is divided into two parts. After calculating a column vector $\vec{y}$ by which elementary row operations are conducted on the basis, the section of the pseudo greatest common divisor (described below) is entered. Since one of the main objectives of the program is to maintain numerical accuracy, the $\vec{y}$ column is divided by a pseudo greatest common divisor. As the values of all the constants are integers a very specialized procedure was devised:

Step 1. $y_k = \min|y_i|, i = 0,\ldots,m+1$

2. Calculate y_k, $y_k/2$, $y_k/3$, $y_k/4$, $y_k/5$

3. Among the constants of step 2 select the largest one which is both an integer and common divisor of all y_i's

4. Divide all entries of $\vec{y}$ by value determined in step 3.

The second section of TRANS transforms all but the last two columns of (13) by the formulae:

$$\begin{aligned} &d_{ij} \text{ replaced by } [d_{ij}y_\ell - d_{\ell i}y_i]\frac{1}{y_\ell} \quad \text{for } i \neq \ell \\ \text{and } &d_{\ell j} \text{ replaced by } d_{\ell j}/y_\ell \qquad\qquad\quad \text{for } i = \ell \end{aligned} \tag{15}$$

where y_ℓ corresponds to min (x_r/y_r) for $y_r > 0$ and $r = 0,\ldots,m+1$. Since the $d_{i,j}$'s and y_i's are all integers, $(d_{ij}y_\ell - y_i d_{\ell j})$ will be integer. If one does not divide by y_ℓ, the entries of (13) will contain

integers. Thus upon completion of every iteration y_ℓ is stored separately and the numerators of (15) are stored in the basis. The division of all the entries of (13) by the product $y_\ell{}^1 y_\ell{}^2 \ldots y_\ell{}^k$ is delayed until the end of Phase II.